Appearing in *International Symposium on Field-Programmable Gate Arrays* (FPGA'03), Feb. 23–25, 2003

Stochastic, Spatial Routing for Hypergraphs, Trees, and Meshes

Randy Huang
Soda Hall #1776
UC Berkeley
Berkeley, CA 94720
rhuang@cs.berkeley.edu

John Wawrzynek
Soda Hall #1776
UC Berkeley
Berkeley, CA 94720
johnw@cs.berkeley.edu

André DeHon
Dept. of CS, 256-80
California Institute of Technology
Pasadena, CA 91125
andre@acm.org

ABSTRACT

FPGA place and route is time consuming, often serving as the major obstacle inhibiting a fast edit-compile-test loop in prototyping and development and the major obstacle preventing late-bound hardware and design mapping for reconfigurable systems. Previous work showed that hardware-assisted routing can accelerate fanout-free routing on Fat-Trees by three orders of magnitude with modest modifications to the network itself. In this paper, we show how these techniques can be applied to any FPGA and how they can be implemented on top of LUT networks in cases where modification of the FPGA itself is not justified. We further show how to accommodate fanout and how to achieve comparable route quality to software-based methods. For a tree network, we estimate an FPGA implementation of our routing logic could route the Toronto Place and Route Benchmarks at least two orders of magnitude faster than a software Pathfinder while achieving within 3% of the aggregate quality. Preliminary results on small mesh benchmarks achieve within one track of `vpr -fast`.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; J.6 [**Computer-Aided Engineering**]: Computer-Aided Design; C.3 [**Special-Purpose and Application-Based Systems**]; C.4 [**Performance of Systems**]: Design Studies

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

FPGA, Detail Routing, Reconfigurable Computing, Spatial Routing

1. INTRODUCTION

The promise of FPGAs has always been rapid-turn around for testing and implementation of new ideas. Unfortunately, FPGA place and route times are now often measured in hours; this slows development and reduces the advantages of a device which can be customized in seconds. *It is the software mapping time, not the device programming time, that often determines how rapidly these devices can be "Field-Programmed."*

Driven by Moore's law semiconductor scaling, we continue to get larger FPGAs, making them suitable for a larger class of applications. However, FPGAs are getting larger faster than conventional computers are getting faster [1]. In addition, with typically superlinear computational requirements for routing, FPGA route time is only increasing as we move into the future.

Can large FPGAs be routed in less than a second? less than a millisecond?

As an alternate to riding the conventional processor curve, we suggest a routing scheme that allows us to ride the FPGA technology curve itself. That is, we use collections of FPGAs, potentially in the same technology as the target FPGA, to route an FPGA. This has two benefits:

1. The parallel, spatial FPGA routing scheme is already substantially faster than software routing.
2. Since these FPGAs ride the same technology curve, we get more FPGA parallelism at exactly the same rate we have larger devices to route.

Depending on the interconnect growth requirements (Rent's Rule), the routing time may still slow down with larger designs, but the time requirements can grow sublinearly.

In previous work [9], we showed that we could augment a Fat-Tree-style network with additional hardware so the device itself could support routing. Routing fanout-free designs, we were able to show three orders of magnitude speedup over state-of-the-art software approaches while sacrificing less than 25% of the quality of the best software approach.

In this paper, we show how we can adapt this solution style so that it has much broader applicability. Our new contributions include:

- Introducing alternative schemes for congestion management which use randomness to avoid Pathfinder's history state. (Section 4)

- Demonstrating that these schemes are highly suitable for spatial hardware implementation and that they achieve comparable quality to the software Pathfinder solution. (Section 4)
- Showing how the hardware designs can accommodate graphs with fanout (hypergraphs). (Section 5)
- Benchmarking the resulting, hardware-based routes against the standard FPGA place and route benchmark suite from Toronto. (Section 5)
- Showing how these ideas can be adapted for more traditional, mesh-based FPGA routing networks; this adaptation suggests sufficient building block techniques to allow this technique to be applied to any network topology. (Section 6)
- Showing how the structure for this router can be mapped into FPGA LUTs so that a large collection of FPGAs could be used to perform the routing of single FPGA. (Section 7)

2. PRIOR WORK

Software Several serious attempts have been made to improve the performance of software-based FPGA routers, including Swartz, Betz, and Rose [15] and Tessier [16]. Both efforts showed similar results, achieving faster routing when allowed to trade quality for time. Greedy, maze routing in Lola is also optimized for router speed at the expense of quality [10], and it achieves similar performance results. In our previous paper, we digested the results from these fast techniques and concluded that they require roughly 75,000-100,000 processor cycles per two-point net.

Multiprocessor Chan and Schlag used a small number of processors (4–5) with modest FPGA-assist to improve FPGA routing times. They were able to show a little over a 2–4× speedup in route time [6] [7].

Hardware Routers In the early 80's, a number of researchers looked at building systolic array hardware to implement Lee's Maze routing algorithm [12] including [5] (also knows as *Pathfinder*) [11] [14] [18]. Our router approach is similar in spirit to these routers. We adapt the connectivity from a simple grid to FPGA switching networks, and we develop schemes for fanout, congestion negotiation, and victimization which go beyond simple least-cost path search.

3. BACKGROUND

HSRA Our tree work builds on the linear switch population HSRA [17] (See Figure 1). A key feature of this network is that the number of switches in each hierarchical switchbox is linear in the number of wires in the switchbox and the total number of switches in the network is linear in the number of endpoints.

This network has an important property which is not shared by Manhattan arrays: There is a unique set of switchboxes between any source and sink. Consequently, global routing is trivial (there is only the one solution), making detail routing our only concern. For our hardware-assisted router, this also means there is a unique "least common ancestor" or "crossover" switchbox between any source and sink. We use this localization to detect route success, or failure, locally in the crossover switchbox. Further, once we select a particular wire (switch) in a crossover switchbox, the path from the crossover to the source and sink, including the set of switches and wires in the path, is completely unique; this property simplifies path identification and allocation. Developing a scheme for path identification is one of the key additions we make to support non-tree networks (Section 6).

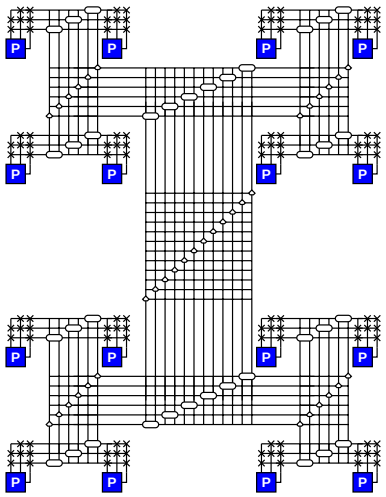

Figure 1: HSRA Network Topology
P's represent the network endpoints (*e.g.* LUTs). The circles and ovals are switchpoints; X's show the connection-box switches. Network shown has 3 base channels.

Pathfinder Pathfinder is the dominant approach to FPGA-routing currently in use in the academic community and heavily used in industry as well. It forms the basis of most of the previous, software-based, attempts to accelerate routing (Section 2). Starting from the base Pathfinder algorithm [13], we implemented our own version for the HSRA [17]. We believe our implementation is very close in spirit to the original. The basic algorithm is as follows:

1. Create a fixed ordering of all nets in the design.
2. While there are unrouted nets and we have not exceeded the maximum number of route trials:
   a. for each net in the original fixed ordering
      - if net is unrouted (no path or shares paths)
        - Perform a *route trial* ≡ rip up the congested net and reroute
   b. update history cost for each congested net

Basic Spatial-Router Scheme The key idea in our FPGA-assisted router is to use the network structure itself (or an analog built on top of a set of FPGAs) to support the parallel route search and to keep track of the state of the network.

To find an available route in the HSRA network, we start at the source and the sink node and trace free (least cost) paths from the source and sink to the crossover switchbox. If the search from the source and the search from the sink meet on one (or more) wires at the crossover switchbox, we have found a viable route path. We can then allocate the path (one of the paths) to this source-sink pair. The scheme is similar for non-tree networks, except that we start our route search only at the source and watch for search completion at the sink.

At the simplest conceptual level, we add a logical OR between the two children channels of each uplink switch in the HSRA and place the OR result on the associated parent channel (See Figure 2). With this addition, we perform a

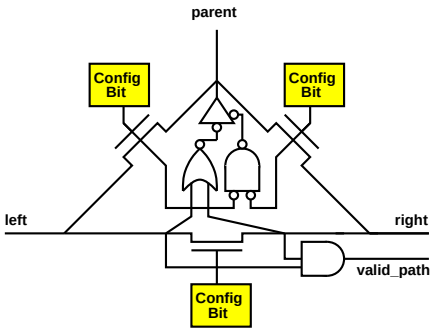

Figure 2: HSRA T-Switch with Path-Search OR

route trial roughly as follows:

1. Set all endpoints (*e.g.* LUTs) to drive zeros into all unused input and output connection to the network and all allocated source lines (leave allocated sink lines undriven as they will be driven by their associated sources).
2. For the designated source-sink pair which we are currently trying to route, drive a one into each unused (available) network connection.
3. Wait for the driven ones to propagate through the network to the unique crossover switchbox; since all the endpoints are driven and all tree switches drive some value upward during this phase (See Figure 2), there are no floating lines in the network.
4. At the crossover switchbox, scan for a switchpoint which receives a one on both of its sibling sides; only this source-sink pair is driving ones, so a matched pair of ones indicates a complete path from both the source and the sink.
5. Allocate the unique path associated with one such matched pair; this means we go ahead and set the switches accordingly to connect this path. Note that this means this path will have zeros driven into it in the future and will not be considered in subsequent route searches.

Figure 3 shows an example of this route search. To route the whole design, we simply perform this search and allocation route trial successively for every connection in the design.

The prospect for acceleration here is simple. In the traditional, software route search, each route trial takes several tens of thousands of cycles (See [9]) to walk a network data structure and to explore all the possible paths between source and sink until a free (or inexpensive) path is found. In this hardware case, we use the network itself to explore all paths simultaneously. It does so quickly because all the switched paths are instantiated in hardware and directly connected by wires. It takes only the signal propagation delay across the wires and switches to trace back all possible paths. If the subsequent allocation can be performed cheaply in place, this turns the whole task from several tens of thousands of cycles into a just a few cycles.

The basic outline sketched here obviously leaves a number of issues open. For the fanout-free case, many of these are detailed in [9]. In the next sections, we address detailed solutions to accommodate fanout and to achieve high quality routes.

4. HIGH QUALITY ROUTING

In this section, we describe several strategies we use to improve the hardware-assisted router quality as measured by the number of tracks needed in the base channel to route a netlist. We start with a description based on fanout-free tree routing. In later sections, we carry these strategies forward to hypergraphs and mesh routing.

Once we have performed a route search for a net, we need a method to select among multiple available paths, or if there is no available path, a way to "free" or victimize a path to route the current net. In our previous work, we described a simple scheme to address these concerns, as well as circuit details on how to perform route search, allocation and victimization in hardware. The simple scheme we proposed is to select a path randomly from multiple free paths if they exist or from all possible paths if they do not.

The main advantage of the random scheme is that it is inexpensive to implement in hardware. However, selecting a victim randomly sometimes produces bad choices, leading the router away from a valid solution or at least causing the router more time to converge. We hypothesize that using some information to bias the selection process will improve the quality of the router and help the router converge more quickly. This leads to several interesting questions:

- What is the right criterion to use in victim selection?
- Does this scheme reduce the number of channels needed to route a netlist?
- How do we implement such scheme in hardware?

4.1 Trading Speed for Quality

Because we use randomness in our algorithm, we get a different result every time. As we noted in our previous work [9], one strategy to improve quality is to leverage the speed improvement gain with hardware assistance and try multiple route starts so that we explore different parts of the route space.

In Table 1, we show the probability distribution function of the random algorithm routing our synthetic benchmarks. We generate a set of difficult synthetic benchmarks to ensure our solution performs reasonably with larger and harder designs; these benchmarks make sure to maximally fill many switchboxes according to the switchbox population, assuring that we can differentiate the effects of route quality and limited population switchboxes. We scale the synthetic network size from 8 to 4,096 and, for each size, we generate 100 netlists which stay unchanged throughout our experiments.

In the table, we show the minimum tracks (W_{min}) achieved from routing each netlist 100 times, the probability distribution of the 10,000 routing tries for each array size, the average, and the expected number of routing tries before achieving W_{min}. We see that it is possible to achieve W_{min} or $W_{min} + 1$ quality within four routing starts. Therefore picking-the-best-of-multiple-starts strategy provides an improvement in routing quality at a slight cost in routing speed.

4.2 Count Congestion

Intuitively, we might expect that the best path to select is the one that does the least damage to existing routes. One way to measure this is to count the number of switches which any new, candidate route shares with existing routes. During a route search, if a switch is occupied, we will increase the cost of the route by one. This strategy is analogous to the Pathfinder algorithm without history ($history = 1$). At the crossover switchbox, we will select a free path ($cost = zero$) if possible or select among the paths least congested.

Comparison We compare the count congestion heuristic with the random heuristic, using the synthetic benchmark

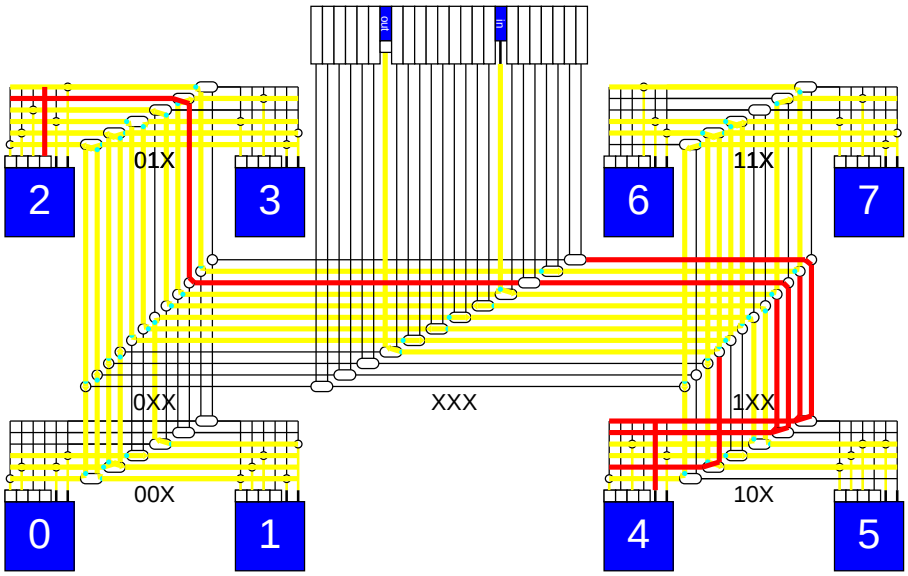

Figure 3: Route Search

Shown here is the result of a path search for a route from node 4 to node 2. The light (yellow), thick lines show pre-existing routes. The dark (red), thick lines show the paths driven to ones by the source and sink and propagated via the up OR logic. At the crossover switchbox (labelled XXX), there is only a single switch which has a one arriving from both sides. We allocate the path that is joined by this switch. Note that there is a single, unique path from the source (node 4) to the sink (node 2) through this switch.

		Random				Count Congestion			
size	W_{min}	$P(W_{min})$	$P(W_{min}+1)$	avg	$E(W_{min})$	$P(W_{min})$	$P(W_{min}+1)$	avg	$E(W_{min})$
8	5	0.966	0.034	5.03	1.0	0.543	0.443	5.47	1.8
16	5	0.273	0.723	5.73	3.7	0.177	0.740	5.91	5.6
32	6	0.505	0.495	6.50	2.0	0.391	0.576	6.64	2.6
64	6	0.000	1.000	7.00	∞	0.086	0.690	7.14	12
128	7	0.441	0.560	7.56	2.3	0.382	0.586	7.65	2.6
256	7	0.000	1.000	8.00	∞	0.068	0.748	8.12	15
512	7	0.000	0.999	8.00	∞	0.002	0.484	8.52	500
1024	8	0.005	0.995	9.00	190	0.124	0.810	8.94	8.1
2048	8	0.000	1.000	9.00	∞	0.010	0.743	9.24	100
4096	9	1.000	0.000	9.00	1.0	0.418	0.568	9.60	2.4

Table 1: Probability Distribution Function for the Random and Count Congestion Algorithms

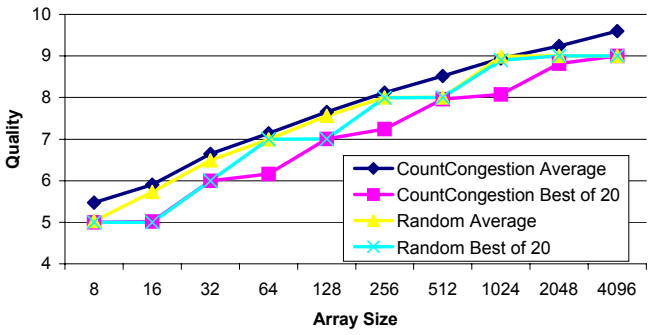

Figure 4: Count Congestion vs. Random Heuristics

Quality is measured by the number of base channel needed to route a netlist, so smaller is better. Array size is the number of LUTs in the array. In our synthetic benchmark, the larger the array, the larger the netlists.

described earlier. We route each netlist 100 times and collect the statistics, shown in Table 1. At first glance, if we just look at the average results, we would conclude that counting congestion is a wasted effort. Across all array sizes, random heuristic *on average* does better than the count congestion heuristic. However, from Table 1, we observe that count congestion heuristic can achieve smaller W_{min}'s and has a higher probability of achieving a better quality route than the random heuristic. This suggests that combining count congestion with the multiple-starts strategy has a chance to improve the quality of the random router.

In Figure 4, we see the combined strategy of multiple-starts and count congestion pays off. Across all array sizes, the combined strategy has improved routing quality by as much as one track! The random heuristic, however, does not benefit as much when combined with multiple starts.

Implementation Count congestion heuristic would not be useful if we were not able to implement it in hardware

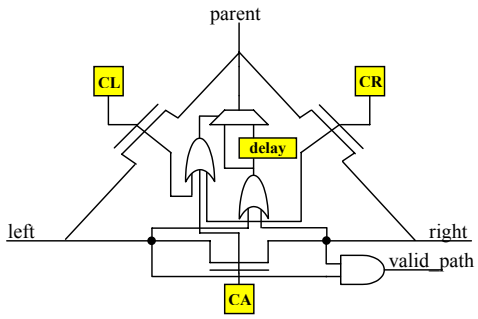

Figure 5: HSRA T-Switch with Congestion Delay

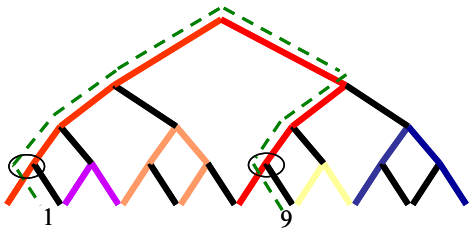

Figure 6: Route search with count net heuristic

efficiently. Fortunately, there is a simple and elegant implementation. If the switch is occupied, we will delay the search signal by one cycle and the first search signal to arrive at the crossover switchbox is the path least congested.

Approximation This scheme does not compute congestion as a sum of the congestion from the source and the sink, which would require an adder at every switch. Instead, we approximate the summation by computing the maximum congestion from the source and the sink. This approximation reduces the required adder into a single AND-gate.

The additional hardware cost is minimal: for every switch, we add a flip flop, a multiplexor and a 3-input OR-gate as shown in Figure 5. The decision to delay the search signal is strictly local by looking at the configuration bits of a switch. As a result this implementation will easily scale as the array size grows.

4.3 Count Net

Instead of counting congestion, another strategy we consider to bias the victim selection is to count the number of nets that would be victimized if a path were selected. In Figure 6, we show one wire channel of a size-sixteen ($p = 0$) tree. Wire segments of the same color are occupied by the same net; unoccupied wire segments have black color. Suppose we were to perform a route search from node 1 to node 9 (shown in green dotted line). Count congestion will return a congestion cost of three while count net will return a cost of one since there is only one net occupying the path. The count net scheme directly reflects the number of existing nets affected by this path and, consequently, the amount of re-routing work that has to be done if a path were chosen to be ripped-up.

Comparison In Figure 7, we plot quality vs. array size for both count cost (congestion and net) strategies combined with multiple starts (best of 20). We include results from Pathfinder as a base-line comparison. For the Pathfinder heuristic, we route each of the 100 netlists once and then average the results for each array size.

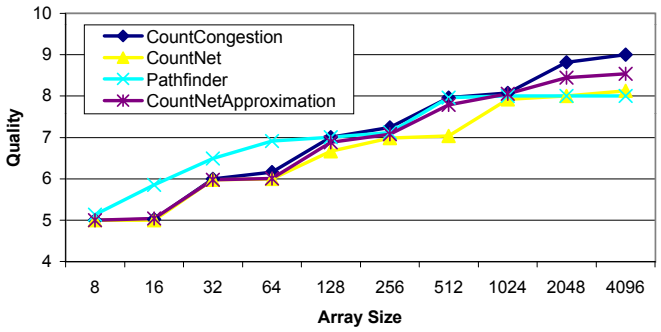

Figure 7: Count cost vs. Pathfinder comparison

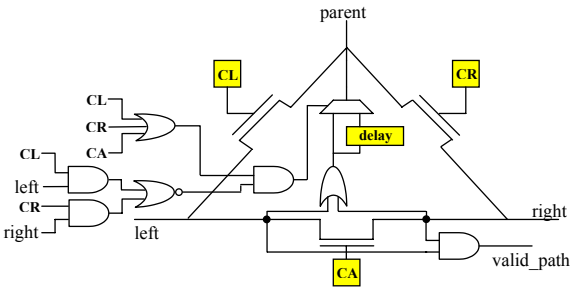

Figure 8: HSRA T-Switch with Count Net Approximation Delay

We see that count net heuristic does better than count congestion heuristic across all array size. In fact, for larger array sizes, we see that count net heuristic (yellow) has closed the gap between the count congestion algorithm (dark blue) and the Pathfinder algorithm (light blue). For smaller array size, count cost (congestion and net) does better than the Pathfinder heuristic because it gets "lucky", exploring route sets which Pathfinder does not; while Pathfinder explores all individual routes, it does not explore all aggregate combinations.

Approximation Although we are encouraged by the fact that a random heuristic can approach the Pathfinder algorithm in routing quality, count net heuristic is costly to implement in hardware; it requires we store a net ID and perform a comparison at every switch.

An inexpensive method to approximate net count heuristic is to observe that a search signal and a routed net can 'interact" at only two switches (entering and existing) as shown in Figure 6 (black circles). At those two switches, the configuration of the switch will be different from the search direction. With additional hardware support, if a switch is occupied and has configuration different from the search direction, we have encounter a new net and will delay the search signal by one cycle. As we can see in Figure 8, the count net approximation implementation requires four additional gates.

Our measurement shows that this scheme will, on average, choose the same net as counting the exact number of nets a path has to victimize 75% of the time. In Figure 7, we see that although the count net approximation heuristic does not approach the quality of count net heuristic, it is consistently better than count congestion heuristic.

5. HYPERGRAPH SUPPORT

In this section, we extend the hardware-assisted algorithm to route netlists with fanouts, describe the necessary modification to the switch, and compare the quality and the routing speed to the Pathfinder algorithm using Toronto benchmark suite.

Routing nets with fanouts The basic ideas behind our scheme is that we sequentially route each two-point net, trying to re-use as much as possible from existing paths allocated to this net. As a result, we avoid the over-constraint and complexity of dealing with multiple sinks simultaneously.

To route nets with fanouts, we add a state bit at every switch. This bit is set when we allocate the switch during the current net search. This bit is cleared when we begin to route a new net. We implement the following simple scheme:

1. Order the destinations associated with a single source by the path length. For a tree network, this is the same as twice the height of the crossover switchbox.
2. For each destination
   - From the sink, we send a search signal on all unused inputs and drive the global route signal. The global route signal is a global binary tree that guides the allocation process. From the crossover switch, during allocation, we need to know which child connection to make to reach the source or the sink; the global-route tree provides this information.
   - From the source, we do nothing and do not drive the global route signal.
   - At a switch, we look at the global route signal to tell us which direction is the sink side; the sink side will have the global route signal driven. The state bit helps us determine if the switch has been allocated during the current net search and therefore can be a point of fanout for the current fanout search. If the state bit is set and the sink side is congestion free, we have found an available path.
   - Otherwise, drive ones into all available source paths and allocate a new path, like a standard route search.

Although this scheme probably uses more resources than optimal, it definitely uses fewer resources than treating each source-sink connection as a separate net.

Approximation In the current count net approximation heuristic, a net with 1,000 fanouts will cost the same as a net with no fanout; if a net with large fanouts is victimized, a large number of two-point nets will be ripped out when the large fanout net is re-routed, resulting in slower convergence and worse routing quality. To deal with this problem, we could count the number of fanouts that would be affected and choose the path with the least fanouts. However, implementing count fanout exactly in hardware appears prohibitively expensive.

We can approximate the count fanout heuristic in a binary fashion with a fanout lock. The idea is that we want to lock down nets with large fanouts after they have been routed and prevent them from being ripped-out. Effectively, this scheme says the cost of a victimizing the high fanout net is infinite so it should not be a victim candidate. Since we order nets by decreasing fanout, high fanout nets will be routed first before they have a chance to interfere with each other. To implement fanout lock in hardware, we:

- Add a lock bit for every switch.
- Assert the lock bit after allocation for high fanout net.

If a switch has an asserted lock-bit, it will not propagate cost signal upward. This assures the crossover switch box will not select a path with high fanout nets. In our implementation, nets with more than ten fanouts are locked after allocation.

Design	LUTs	Fanout Lock (by # starts)			Pathfnd
		5	10	20	RT=50
alu4	1522	10.00	10.00	10.00	10
apex2	1878	10.59	10.35	10.12	11
apex4	1262	11.00	11.00	11.00	11
bigkey	1707	8.27	8.07	8.01	9
clma	8383	11.00	11.00	11.00	11
des	1591	10.00	10.00	10.00	9
diffeq	1497	9.96	9.92	9.84	8
dsip	1370	8.57	8.32	8.10	9
elliptic	8192	10.04	10.00	10.00	10
ex1010	4598	13.22	12.97	12.80	10
ex5p	1064	11.00	11.00	11.00	10
frisc	3556	10.82	10.67	10.45	10
misex3	1397	10.21	10.04	10.00	11
pdc	4575	12.01	12.00	12.00	12
s298	1931	9.49	9.24	9.06	9
s38417	6406	10.00	10.00	10.00	9
s38584.1	6446	9.00	9.00	9.00	9
seq	1750	10.05	10.00	10.00	11
spla	3690	12.57	12.33	12.11	12
tseng	1047	10.00	10.00	10.00	8
Total		207.81	205.92	204.50	199

Table 2: Route Quality Comparison

Quality Comparison For comparison, we use the standard FPGA place and route benchmark suite from Toronto [4]. The benchmark suite is placed on a tree network using tools we developed for the HSRA [8]. The growth rate of the network is governed by the Rent's parameter (p) and is selected to be 0.6 based on our previous work. The same placement is used for both routing algorithms and thus the comparison is fair regardless of the placement tool.

For the fanout lock algorithm, we route each netlist 500 times and use the statistics to calculate the expected number of tracks if we were to pick the best of multiple starts. For the Pathfinder algorithm, we set the route trial (RT) multiplier to be 50, which represents a high routing effort; this means we allow the router to attempt a number of individual, two-point net route trials equal to 50× the total number two-point nets in the design.

In Table 2, we list the quality results of routing each benchmark using both algorithms. We see that as we increase the number of route starts, the chance of find better quality route increases as well. More importantly, the results show that a random algorithm simple enough to be implemented in hardware can approach the Pathfinder algorithm in terms of quality (within 3%).

Routing speed comparison To measure running time of the software router, we use the 64b TSC (Time Stamp Counter) timer on the processor to measure time in cycles. Our software router is written in C and compiled with the GNU C Compiler (version 2.95.2) using `-o3` option. Benchmarks are run on 1.4 GHz Pentium3-based system running Linux 2.4.18 with 133MHz system bus and variable amount of main memory (from 256MB to 2GB). In general, since we are performing graph traversals on a large data struc-

Design	W_{min}	Software Pathfinder Cycles(Millions) W_{min}	$W_{min}+1$	$W_{min}+2$	cyc/2pt net @$W_{min}+2$	Hardware Route Cycles W_{min}	$W_{min}+1$	$W_{min}+2$	Adjusted Speedup
alu4	10	13,173	799	769	142K	281,124	126,471	123,031	4,686
apex2	10	N/A	676	529	79K	1,339,577	184,100	129,086	367
apex4	11	12,256	417	389	87K	167,262	147,580	147,009	7,328
bigkey	8	N/A	11,146	931	142K	2,197,956	111,011	99,991	10,041
clma	11	42,061	11,614	11,481	169K	2,418,400	562,298	554,284	1,739
des	9	95,352	370	356	58K	N/A	150,267	131,689	246
diffeq	8	367,254	49,284	486	90K	N/A	13,025,311	99,296	378
dsip	8	N/A	4,717	1,662	285K	1,024,711	106,747	95,013	4,419
elliptic	10	924,960	3,538	2,224	182K	694,776	277,612	272,066	133,131
ex1010	10	1,529,242	318,979	62,709	292K	N/A	N/A	29,727,647	211
ex5p	10	121,369	4,540	375	94K	N/A	259,207	143,262	1,752
frisc	10	1,765,744	3,149	2,036	164K	7,583,946	286,291	220,987	23,283
misex3	10	N/A	477	493	93K	510,111	132,005	100,865	361
pdc	12	24,552	2,865	2,810	163K	989,232	544,587	540,704	2,482
s298	9	576,848	1,762	1,332	192K	1,190,185	146,285	134,887	48,467
s38417	9	489,715	8,231	2,874	141K	N/A	221,047	184,183	3,723
s38584	9	215,098	3,422	3,283	159K	410,546	224,569	218,786	52,393
seq	10	N/A	620	598	96K	618,319	183,538	175,087	338
spla	12	14,248	1,750	1,650	116K	3,710,998	512,716	375,272	384
tseng	8	173,711	23,566	385	97K	N/A	N/A	88,150	436

Table 3: Pathfinder and Fanout Lock Route Time Comparison

ture, most memory accesses will be cache misses. However, we make sure our entire data structure will fit in the main memory, so that we are not measuring performance derated by virtual memory thrashing.

For the software Pathfinder algorithm, we run each benchmark three times and report the minimum time. In Table 3, we list the minimum number of track needed to route each benchmark (W_{min}), the routing time for W_{min}, $W_{min}+1$, and $W_{min}+2$, and average number of CPU cycles needed to route a 2-point net (for $W_{min}+2$). An "N/A" entry indicates that the router fails to route at that given number of tracks. Our implementation of the Pathfinder algorithm averages 140K cycles per 2-point net, which is comparable with previous work.

For the fanout lock algorithm, we measure the total number of route search, rip-outs, and fanout search from our software implementation. With these three parameters, we calculate the speedup with hardware assistance similar to the calculation performed in our previous work. We list the **expected** routing time for W_{min}, $W_{min}+1$, and $W_{min}+2$ in the table. We see that fanout lock is able to achieve equal or better quality results on fourteen of the benchmarks. Finally, assuming a 10× difference in clock rates between the CPU and the FPGA (*e.g.* 6 ns FPGA cycle and 0.6 ns CPU cycle), we are still able to achieve speedups of over two orders of magnitude. Summing the minimum channel requirements across all 20 benchmarks, software Pathfinder requires 199 channels, whereas the fanout lock algorithm requires 202.

6. MESH ROUTING

The idea of doing a direct path search on the hardware or an analog thereof is applicable to any network topology. In general, however:

- There is no well defined crossover point which will contain all possible routes.
- It is not obvious which direction through a switchpoint actually leads to the shortest path to the destination.
- The path back to the source is not implied directly by the topology of the routing network.
- Not all paths from source to sink are the same length and non-minimal length paths may be important components of a good solution.

The more general hardware-search strategy is to start a path search as before with the source driving a one into its output and all non-sources driving zeros. In this case, we do not drive from the sink. Rather, we have the sink listen for the arrival of a one on one of its inputs. The switches are designed to propagate the one along any free path in the network without delay and to propagate along congested paths only after inserting an appropriate delay to approximate an appropriate congestion delay. Using this basic scheme, the signal from the least delay path, hopefully least congested, will arrive at the destination first.

Back Paths The key new problem in this case is: *how do we find the path back to the source and negotiate among equivalent, alternative paths?*

Borrowing from Hansel and Gretel, we leave "breadcrumbs" to mark our path back from the source to the sink. That is, each switchpoint notes which input arrives first and marks that input as the appropriate direction to route an allocate signal should it subsequently receive one. It is quite possible, and quite likely, that two or more search signals arrive at the same switchpoint at the same time. In the example of a traditional, diamond-switchpoint in a mesh, a search emanating from a LUT in the lower Southwest portion of a chip can easily deliver search inputs to a switch to its northeast on both the South and West inputs simultaneously. To promote stochastic path selection as we found beneficial in the tree case, we allow the switchpoint to select randomly among the input signals arriving at the same time. when multiple signals arrive in a destination C-Box (or any crossbar like structure), we select randomly among all the equivalent sources. We call the selected input the **preferred** input. Here, unlike the tree, we distribute our

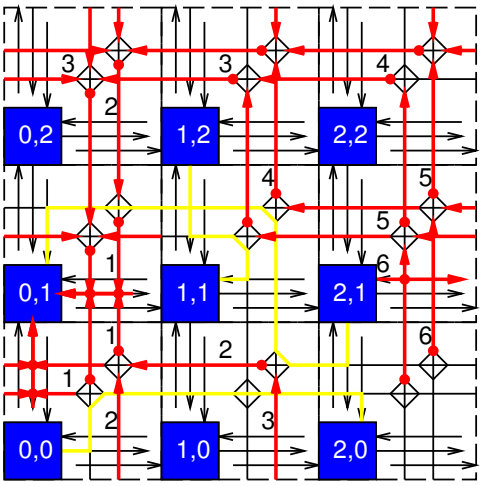

Figure 9: Mesh Route Search

Shown here is the result of a path search for a route from node (0,1) to node (2,1). The light (yellow), thick lines show pre-existing routes. The dark (red), thick lines show the paths driven to ones by the source and propagated in the search; the dots indicate the **preferred** path and arrows show the backpath direction. The numbers mark the timestep when the search signal reaches the annotated channel. Note the switchpoints at (2,1) and (0,0) receive inputs form two different directions on the same timestep.

random selection along the path instead of making a single random selection at the end. Figure 9 shows a sample route search.

Allocation proceeds analogous to the tree case. We drive a one into the selected input at the sink. This one will follow the stored preferences back to the source, marking the switchpoints which the path touches as allocation choices. As before, if this new path intersects with an existing path, the switches are marked as victims. A victim identification phase allows all victim paths to be identified and dropped from the network. The source records the fact that it was victimized so the route controller will know that it needs to be rerouted.

Fanout To support fanout in the mesh, we route all of the destinations (two-point connections in a net) one at a time in sequence and add additional state to track the switchpoints which are allocated by the current net. To attempt to minimize the resources used by each net, we allow path search to flow along paths already allocated to this net. The basic path search for each endpoints is as follows:

1. Drive a one into the source and allow it to propagate along the already allocated path.
2. Continue search, allowing the search to proceed outward from the existing path through free paths, but do not allow any signal propagation through congested paths; this has the effect of find the shortest, congestion-free extension of the existing net, if there is one.
3. If that fails, start a fresh search back at the sources, but keep the path preferences from the previous search where appropriate; this new search from the source makes sure that we find the shortest path according to the standard congestion delay metric to the sink. The new path may be routed on a new track if that is the least cost path. Since we keep the preferences from the previous search, existing paths will always be preferred over new paths when they are the same length; however if a new path is shorter, which can happen because of victimization, the shorter path is taken. Had we not restarted this victimizing search from the source, we could not guarantee to find the path with the least victimization.

Variations There are many variations on this scheme as with the tree scheme. Notably:

- **atomic victimization**—we can remove nets either atomically or one link at a time; the simplest scheme for non-atomic victimization is to simply victimize the net atomically, then add back all the destinations which can be added without victimizing existing paths; this requires that we take time to clear out a net (drop it) before we reroute it.
- **count net transitions**—by allowing path search forward along previously allocated paths without delay, we can count the number of nets which intersect a path rather than the number of used switches (See Section 4.3).
- **congestion delay**—since the mesh (and networks in general) may have non-minimal length paths, simply delaying the search signal one cycle is not adequate to distinguish between a congested route and a longer, uncongested route. Consequently, we consider increased congestion delays to help mitigate the aliasing effects.

Experiments To evaluate the viability of this scheme, we implemented a low-level simulator of the routing logic described in Section 7.2. Since this was a direct simulation of the circuits, the uniprocessor simulation is quite slow preventing us from running large benchmarks. Nonetheless, to provide an initial characterization, we took six LUT-mapped MCNC benchmarks which would fit onto a 10×10 array and routed them both with this FPGA-based mesh routing algorithm and with VPR [3].

We used the `vpr422_challenge_arch` architecture as the mesh architecture for routing; this has single-length segments, a single LUT per island, and uses a subset (diamond) switch; each of the 4 LUT inputs appears on a single side of the logic block ($T_{in} = 1$), and the output appears on two sides ($T_{out} = 2$); both are fully populated ($F_c = 1$). We use VPR 4.3 [2] to place the designs for **both** routers. We use the channel minimizing VPR 4.3 router in both quality and fast mode as our quality comparison. VPR was compiled `-o3` and run on a P3-1.4 GHz computer for timing.

Results Table 4 summarizes our routing results. With minimal effort and no randomness, the spatial router gets within 1 or 2 tracks of the VPR results. The deterministic router can get itself into bad victimization loops as the non-atomic case demonstrates on 5xp1. With random path selection and non-atomic victimization, the spatial router achieves the same quality as the VPR fast routing scheme in 4 of the 6 cases.

Table 5 summarizes the cycles and runtime for both VPR and the spatial router. For the VPR results, we ran VPR in the fixed channel width target mode and recorded only the cycles for the actual route using the Pentium TSC counter. We ran the VPR routes three times, checked that they were reasonably consistent from run to run, and recorded the least cycles across the runs. For the spatial route, the cycle counts come directly from the low-level, cycle-accurate simulator and are the primary metric showing the time and effort required to find routes; in order to provide absolute time estimates for comparison, we assume an FPGA implementation

Design	LUTs	VPR 4.3		FPGA-based Router (congestion delay=10)					
		quality	fast	deterministic		rnd atomic		rnd, not atomic	
				atomic	not		cnt trns		cnt trns
5xp1	83	4	5	6	–	6	6	5	6
c8	87	5	5	7	7	6	6	6	6
ex2	90	5	6	6	6	7	6	6	6
mm9a	96	5	5	6	6	6	6	6	6
s526	84	5	5	6	6	6	6	5	5
s526n	90	4	5	6	6	6	5	5	5

Table 4: Mesh Quality Results

of the search logic with a 5 ns cycle. Easy routes show over 250× acceleration. Challenging routes still achieve greater than 40× acceleration.

7. RESOURCE REQUIREMENTS

7.1 Tree Implementation

Implementing the logic equations for a tree switchpoint (*e.g.* Figure 8) in LUTs will require around 21 LUTs, two of which are added for fanout support. Moreover, we need to consider the additional logic needed at the switch box level such as the random number generator, the parallel prefix circuit, and various control signals between the switch box and the global route controller. We conservatively estimate that it would require 9 more LUTs per switch for the switch-box level logic. As a result, it would require 30 LUTs to simulate a fast-routing switch.

Conservatively counting a π-switch (two up-links) as two T-switches (one uplink, shown in Figure 2) and assuming $p = 0.67$, the total number of switchpoints in a design will be $5 \cdot N_{array} \cdot C$. This is easily seen by observing that there are $\frac{N_{array}}{2}$ tree switchboxes at the lowest level of the tree, each of which hold C π-switches. One level up, we have $\frac{N_{array}}{4}$ switchboxes with $2 \cdot C$ π-switches. At the second level up we have $\frac{N_{array}}{4}$ T-switchbox which holds $2 \cdot C$ T-switches. For $p = 0.67$, this pattern of π- and T-switchboxes then repeats, so that we have:

$$N_{sw} = 5 \cdot N_{array} \cdot C \quad (1)$$

In general, the switch constant, which is 5 here, depends on the particular p value ($0.5 < p < 1.0$). Combining with the 30 LUTs per T-switch equivalent, we see we need about 150 LUTs per base channel per leaf tree LUT. For $C = 13$ (largest value in Table 2), this means the network will have 65 T-switch equivalents switches per leaf tree LUT, or around 2000 LUTs per leaf tree LUT. Some additional logic will be required to support the base channel. Since base channels are independent, the logic can easily be sequentialized by base channel, allowing us to save up to a factor of C in LUT count. The time cost will be less than a factor of C since many operations (*e.g.* allocation, victimization, and fanout-free route extensions) need to be performed only on a single base channel tree at a time.

7.2 Mesh Implementation

Table 6 shows that we need around 160 4-LUTs per switchpoint to implement the switch routing logic for the mesh; about half of these LUTs are required to support fanout. Note that features like the SRL16 [20] [19] allow us to implement large congestion delays in conventional FPGAs without a large cost. We will need about this much logic again per track to support the connection box (C-box). As a result we need several hundred LUTs per track. Since tracks are independent, it would likely be benefitial to sequentialize track search so that we contain the total design to several hundred LUTs per LUT in the original design.

Logical Function	Number 4-input LEs	Total LEs per SwitchPoints
Outs	12×4+5	53
Congest	1×4	4
Allocate	6×2	12
Net Share	6×3	18
Victim	6×4+2	26
Config	6×2	12
Prefer	30	30
Total		155

Table 6: Diamond Switchpoint Resource Summary

8. SUMMARY

We have shown that it is possible to design a spatial routing structure that achieves comparable quality to Pathfinder, the state-of-the-art software routing scheme. The spatial router can accommodate fanout and can be adapted for less regular topologies than trees, such as meshes. Supporting history costs directly in the spatial structure appears prohibitively expensive; however, suitable application of random path selection and route locking appears to be an adequate substitute for Pathfinder's history records. Even with multiple route re-starts needed to stochastically explore the route space, the parallel, spatial structure can find routes in three to six orders of magnitude fewer cycles than the sequential, software routers. If we must derate this by an order of magnitude to account for the spatial transit time between switch logic in an FPGA implementation, an FPGA-based spatial router can still be two to five orders of magnitude faster than the software router. This is sufficient to place many routing tasks in the millisecond or sub-millisecond range.

9. ACKNOWLEDGMENTS

This research was funded in part by the DARPA Moletronics program under grant ONR N00014-01-0651 and by the NSF CAREER program under grant CCR-0133102.

10. REFERENCES

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings fo the 27th International Symposium on Computer Architecture*, pages 248–259, 2000.

Design	LUTs	VPR 4.3 `-fast` cycles (time) on P3-1.4GHz w=7	w=6	w=5
5xp1	83	51038890 (36ms)	106681978 (76ms)	72812545 (52ms)
c8	87	64300134 (46ms)	44116126 (31ms)	90248704 (64ms)
ex2	90	86430391 (62ms)	105175532 (75ms)	not route
mm9a	96	25854713 (18ms)	49484479 (35ms)	121929509 (87ms)
s526	84	34459634 (24ms)	53019756 (37ms)	85979580 (61ms)
s526n	90	23618103 (17ms)	54809964 (39ms)	99866126 (71ms)
Design	**LUTs**	FPGA-based Router (assume 5ns cycle) w=7	w=6	w=5
5xp1	83	28333 (140μs)	24319 (120μs)	160777 (800μs)
c8	87	17383 (87μs)	29660 (150μs)	not route
ex2	90	15029 (75μs)	15997 (80μs)	not route
mm9a	96	16727 (84μs)	28083 (140μs)	not route
s526	84	11050 (55μs)	11054 (55μs)	295222 (1.5ms)
s526n	90	11369 (57μs)	11369 (57μs)	177475 (890μs)

Table 5: Mesh Route Time Results

[2] V. Betz. VPR and T-VPack: Versatile Packing, Placement and Routing for FPGAs. <http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html>, March 27 1999. Version 4.30.

[3] V. Betz and J. Rose. VPR: A New Packing, Placement, and Routing Tool for FPGA Research. In W. Luk, P. Y. K. Cheung, and M. Glesner, editors, *Field-Programmable Logic and Applications*, number 1304 in LNCS, pages 213–222. Springer, August 1997.

[4] V. Betz and J. Rose. FPGA Place-and-Route Challenge. <http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html>, 1999.

[5] C. R. Carroll. A Smart Memory Array Processor for Two Layer Path Finding. In *Proceedings of the Second Caltech Conference on Very Large Scale Integration*, pages 165–195, January 1981.

[6] P. K. Chan and M. D. F. Schlag. Acceleration of an FPGA Router. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–181. IEEE, April 1997.

[7] P. K. Chan and M. D. F. Schlag. New Parallelization and Convergence Results for NC: A Negotiation-Based FPGA Router. In *Proceedings of the 2000 International Symposium on Field-Programmable Gate Arrays (FPGA'00)*, pages 165–174. ACM/SIGDA, February 2000.

[8] A. DeHon. Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization). In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 69–78, February 1999.

[9] A. DeHon, R. Huang, and J. Wawrzynek. Hardware-Assisted Fast Routing. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 205–215, April 2002.

[10] S. W. Gehring and S. H.-M. Ludwig. Fast Integrated Tools for Circuit Design with FPGAs. In *Proceedings of the 1998 International Symposium on Field-Programmable Gate Arrays (FPGA'98)*, pages 133–139. ACM/SIGDA, February 1998.

[11] A. Iosupovici. A Class of Array Architectures for Hardware Grid Routers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 5(2):245–255, April 1986.

[12] C. Y. Lee. An Algorithm for Path Connectios and Its Applications. *IRE Transactions on Electronic Computers*, EC-10:346–365, 1961.

[13] L. McMurchie and C. Ebling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 111–117. ACM, February 1995.

[14] T. Ryan and E. Rogers. An ISMA Lee Router Accelerator. *IEEE Design and Test of Computers*, pages 38–45, October 1987.

[15] J. S. Swarz, V. Betz, and J. Rose. A Fast Routability-Driven Router for FPGAs. In *Proceedings of the 1998 International Symposium on Field-Programmable Gate Arrays (FPGA'98)*, pages 140–149. ACM/SIGDA, February 1998.

[16] R. Tessier. Negotiated A* Routing for FPGAs. In *Proceedings of the 5th Canadian Workshop on Field Programmable Devices*, June 1998.

[17] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon. HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 125–134, February 1999.

[18] T. Watanabe, H. Kitazawa, and Y. Sugiyama. A Parallel Adaptable Routing Algorithm and its Implementation on a Two-Dimensional Array Processor. *IEEE Transactions on Computer-Aided Design*, 6(2):241–250, March 1987.

[19] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Linear Feedback Shift Registers in Virtex Devices*, January 2001. XAPP 210 <http://www.xilinx.com/xapp/xapp210.pdf>.

[20] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Xilinx Virtex-II 1.5V Platform FPGAs Data Sheet* , July 2002. DS031 <http://www.xilinx.com/partinfo/ds031.pdf>.

APPENDIX

A. DETAILED TREE ROUTE STATISTICS

For the fanout lock algorithm, we measure the total number of route search, rip-outs, and fanout search from our software implementation (Table 7). These were used to calculate the speedups reported in Table 3.

Design	#LUTs	Size	#Nets	N_{rt}	N_{ro}	N_{fo}
alu4	1522	4096	1536	1925	192	5712
apex2	1878	4096	1916	2357	106	6907
apex4	1262	4096	1269	1642	96	4685
bigkey	1707	2048	1935	2324	130	6569
des	1591	2048	1847	2236	115	6298
diffeq	1497	2048	1560	2065	264	5641
dsip	1370	2048	1598	1955	139	5887
elliptic	8192	8192	3734	4343	178	12455
ex1010	4598	8192	4608	5214	14	16086
ex5p	1064	4096	1072	1759	400	4663
frisc	3556	8192	3575	4425	274	12849
misex3	1397	4096	1411	1769	99	5164
pdc	4575	16384	4591	5587	122	17379
s298	1931	4096	1934	2175	206	7166
s38417	6406	8192	6434	8380	956	22185
s38584.1	6446	8192	6483	7041	47	20657
seq	1750	4096	1791	2202	94	6367
spla	3690	16384	3706	4346	80	13951
tseng	1047	2048	1098	1454	174	3920

Table 7: Statistics from Fanout Lock Router on Toronto Benchmark Set

Size indicates the size of the physical tree on which the benchmark is placed; these are all powers of two and may be depopulated to match the $p = 0.6$ growth rate [8]. N_{rt} is the number of route search, N_{ro} is the number of victimization, and N_{fo} is the number of fanout search.

B. SWITCH LOGIC

Figure 10 presents the logic equations needed to implement a tree switch that supports hardware-assisted routing. Figure 11 shows a representative set of equations for the diamond switchpoint. Table 8 is an expanded version of Table 6 which summarizes the resources requirements for the diamond switchpoint.

Equation 1 is easily derived by noticing that summing the the switches contributed per level to the endpoint forms a geometric sum:

$$\begin{aligned} N_{sw} &= N_{array} \cdot \left(\left(\frac{2 \cdot C}{2} + \frac{2 \cdot 2 \cdot C}{4} + \frac{1 \cdot 4 \cdot C}{8} \right) + \right. \\ &\qquad \left. \left(\frac{2 \cdot 4 \cdot C}{16} + \frac{2 \cdot 8 \cdot C}{32} + \frac{1 \cdot 16 \cdot C}{64} \right) + \ldots \right) \\ &= N_{array} \cdot C \cdot \left(\left(1 + 1 + \frac{1}{2} \right) + \left(\frac{1}{2} + \frac{1}{2} + \frac{1}{4} \right) + \ldots \right) \\ &= N_{array} \cdot C \cdot \left(\frac{5}{2} + \frac{5}{4} + \frac{5}{8} + \ldots \right) \\ &\leq N_{array} \cdot C \cdot \left(\frac{5}{2} \cdot 2 \right) = 5 \cdot N_{array} \cdot C \end{aligned}$$

Logical Function	Number 4-input LEs	Total LEs per SwitchPoints
Outs	12×4+5	53
Congest	1×4	4
Allocate	6×2	12
ATN	6×1	6
UON	6×2	12
Victim	6×4+2	26
Config	6×2	12
Prefer	30	30
Total		155

Table 8: Diamond Switchpoint Resource Summary

$$
\begin{aligned}
VictimLatch &:= ((Parent_{in} \cdot IdentifyVictim \cdot (CL + CR)) + (Parent_{in} \cdot DropBit)) \cdot \overline{ClearVictim} \\
ConfigClear &= Drop \cdot VictimLatch \\
CL &:= GlobalRoute_{left} \cdot Allocate \cdot Parent_{in} \cdot \overline{ConfigClear} \\
CR &:= GlobalRoute_{right} \cdot Allocate \cdot Parent_{in} \cdot \overline{ConfigClear} \\
CA &:= AllocateThisPath \cdot Allocate \\
Switch_{left} &= CL + (IdentifyVictim \cdot GlobalRoute_{left}) \\
Switch_{right} &= CR + (IdentifyVictim \cdot GlobalRoute_{right}) \\
Switch_{across} &= CA \\
Occupied &= CA + CR + CL \\
Delay &= Occupied \cdot \overline{((Left_{in} \cdot CL) + (Right_{in} \cdot CR))} \\
LeftOrRight &= Left_{in} + Right_{in} \\
LeftOrRightFF &:= LeftOrRight \\
LockFF &= Lock \cdot Parent_{out} \\
Search &= \overline{LockFF} \cdot (Delay \cdot LeftOrRightFF + \overline{Delay} \cdot LeftOrRight) \\
Left_{out} &= AllocateThisPath + (Switch_{across} \cdot Right_{in}) + (Switch_{left} \cdot Parent_{in}) \\
Right_{out} &= AllocateThisPath + (Switch_{across} \cdot Left_{in}) + (Switch_{right} \cdot Parent_{in}) \\
Parent_{out} &= (VictimSearch \cdot VictimLatch) + Search + (Switch_{left} \cdot Left_{in}) + (Switch_{right} \cdot Right_{in}) \\
FanoutFF &:= Allocate \cdot (AllocateThisPath + Parent_{in}) \cdot \overline{NewNet} \\
ValidPath &= (Left_{in} \cdot Right_{in}) + (FanoutFF \cdot ((Left_{in} \cdot GlobaRoute_{left}) + (Right_{in} \cdot GlobalRoute_{right})) \\
VictimPath &= Left_{in} + Right_{in}
\end{aligned}
$$

Figure 10: Logic Equations for a T-switchpoint

CL, CR, and CA are the configuration bits for the left, right, and across transistors. $GlobalRoute_{left}$ and $GlobalRoute_{right}$ are part of global route tree that guides the allocation process. Allocate, VictimSearch, IdentifyVictim, ClearVictim, Drop, DropBit, Lock, and NewNet are global control signals that help guide the routing process.

$$
\begin{aligned}
AnyIn &= N_{in} + E_{in} + S_{in} + W_{in} \\
AnyAlloc &= NS_{alloc} + NE_{alloc} + NW_{alloc} + SE_{alloc} + SW_{alloc} + EW_{alloc} \\
N_{out\_calc} &= Search \cdot \overline{N_{in}} \cdot \overline{N_{uon}} \cdot \left(S_{in} \cdot \overline{S_{uon}} + W_{in} \cdot \overline{W_{uon}} + E_{in} \cdot \overline{E_{uon}}\right) \cdot \overline{Prefer[2:0] = N} \\
&\quad + Allocate \cdot AnyIn \cdot (Prefer[2:0] = N) \\
&\quad + IdentifyVictim \cdot (NS_{victim} + NW_{victim} + NE_{victim}) \\
&\quad + (Search + SensitizePath) \cdot (E_{in} \cdot NE_{atn} + S_{in} \cdot NS_{atn} + W_{in} \cdot NW_{atn}) \cdot \overline{Prefer[2:0] = N} \\
&\quad + N_{out\_congest} \cdot \overline{SensitizePath} \cdot \overline{Prefer[2:0] = N} \\
N_{out\_congest} &:= Search \cdot (E_{in} + S_{in} + W_{in}) \\
N_{out} &:= \overline{SearchBegin} \cdot N_{out\_calc} + SearchBegin \cdot N_{out} \\
NS &:= (Install \cdot NS_{alloc} + NS) \cdot \overline{\left(Clear + Install \cdot NS_{victim} \cdot \overline{NS_{alloc}}\right)} \\
NS_{atn} &:= \overline{SearchNetBegin} \cdot (NS_{atn} + NS_{alloc} \cdot Install) \\
N_{uon} &= NS \cdot \overline{NS_{atn}} + NE \cdot \overline{NE_{atn}} + NW \cdot \overline{NW_{atn}} \\
NS_{alloc} &:= Allocate \cdot (N_{in} \cdot (Prefer[2:0] == S) + S_{in} \cdot (Prefer[2:0] == N)) + \overline{SearchBegin} \cdot NS_{alloc} \\
NS_{victim} &:= \overline{SearchBegin} \cdot NS \cdot \overline{NS_{atn}} \cdot \left(AnyAlloc \cdot \overline{EW_{alloc}} + IdentifyVictim \cdot (N_{in} + S_{in})\right) \\
&\quad + \overline{SearchBegin} \cdot NS_{victim}
\end{aligned}
$$

N.B. uon designates "Used by Other Nets"; atn designates "Allocated for This Net".

Figure 11: Representative Set of Logic Equation for Diamond Switchpoint